**INFORMATION SYSTEMS SERVICES**

# Further Exercises in Unix

***This document contains a series of exercises which provide examples of Unix commands and is intended for existing Unix users***

**AUTHOR:** Information Systems Services

**DATE:** August 2004

**EDITION:** 1.9

**TUT 14**

**50p**

# Contents



**Format Conventions**

In this document the following format conventions are used:

| | |
|---|---|
| Commands that you must type in are shown in **`bold Courier`** font. | **`WIN31`** |
| Menu items are given in a **Bold, Arial** font. | **Windows Applications** |
| Keys that you press are enclosed in angle brackets. | **<Enter>** |

**Feedback**

If you notice any mistakes in this document please contact the Information Officer. Email should be sent to the address **`info-officer@leeds.ac.uk`**




**Print Record**

This document was printed on 13-Sep-04.

# Aim of this Document

This document contains a series of exercises which are intended to provide the Unix user with hands-on experience in the use of selected Unix commands. It is assumed that you are familiar with the material covered in the document *Introduction To Unix: Exercises* (TUT 5) and *Getting Started With Unix* (BEG 8) and that you can use an editor to create files on the system being used.

This document introduces the following commands:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **set** | **csh** | **echo** | **who** | **wc** | **printenv** | **history** | **clear** |
| **alias** | **unalias** | **chmode** | **ps** | **fg** | **jobs** | **kill** | **bg** |
| | **stop** | | | | | | |

Further information on these commands can be obtained by typing **man** *command*.

This document should be used while logged in to a Unix system. The exercises can be attempted either on your own, or as part of a class. You **must** have a user name on a Unix system in order to try the exercises.

# Task 1 Identifying The Unix Shell

**Objective** To check that you are working under the correct shell for this set of exercises.

**Comments** The *Unix shell* is the command language interpreter through which the user communicates with the operating system and hence is able to carry out actions on the system. There are several different shells available for Unix systems (including the Bourne, C and Korn shells). For these exercises you should be using the *C shell* which is the default on Unix systems managed by Information Systems Services.

---

**Activity 1.1** Login to your Unix system and type the command:

```
% set
```

The system should return a list of *shell variables* that will include a line similar to:

```
shell /bin/csh
```

or

```
SHELL=/bin/sh
```

If the given shell is `/bin/csh` then you are working under the C shell (csh);

if it is `/bin/sh` you are working under the Bourne shell (sh).

**Activity 1.2** If you have found out that your system is running a shell other than the C shell type the command:

```
% csh
```
[1]

This will change your shell to the C shell. The change will remain in effect until you log out.

**Note** You will normally find that the C shell prompt is of the format `hostname%`; for example, `sun%`. You should type commands next to the shell prompt.

---

[1] If you type **`csh`** while working under the C shell, you will create another C shell within your original one. When you come to logout you will get the message `Not Login Shell`. To exit this shell type `<Ctrl D>`, which will return you to the original shell. You can then logout normally.

# Task 2 Using The Unix Shell

**Objective** To use simple shell commands.

**Comments** In these exercises it is assumed that you are working under the C shell.

---

**Activity 2.1** Type in the command:

```
% car /etc/hosts
```

The Unix shell will evaluate the command line. In this case the shell will display the message:

car: command not found

The shell looked for a command **car** but since this does not exist an error message was returned. The correct command should have been **cat**

**Activity 2.2** The shell can perform variable substitution. Type in:

```
% echo "Home Directory Is $home"
```

The shell will reply with:

Home Directory Is /home/gps/men5jb

The shell variable `$home` has been evaluated and displayed.

**Activity 2.3** The shell controls piping the output from one command to the input of another command using the pipe operator. Type the command:

```
% who | wc
```

The shell will reply with three numbers. The output from the **who** command (a list of users logged in), has been piped to **wc** (the word count program), which returns the number of lines, words and characters read.

**Activity 2.4** The shell controls redirection of the output from commands into files. Type the command:

```
% who > logged.users
```

Then look at the contents of the file by giving the command:

```
% more logged.users
```

# Task 3 Shell Variables

**Objective** To find out the shell variables that have been set at login time, to find out how to access these variables and how to alter them.

**Comments** Some shell variables are read only so you will be unable to alter them.

---

**Activity 3.1** Type the **set** command to see which variables have been set for you by the system:

```
% set
```

You should get a list returned that will include some lines similar to the following:

```
home     /home/gps/men5jb
shell    /bin/csh
user     men5jb
term     vt100
history 30
```

The variable home, shell, etc. are *shell variable names* that have been assigned values at login time. The value of a shell variable may be accessed by $*variable_name* For example type the command:

```
% echo $user $home
```

```
% ls $home
```

For user men5jb the the shell will expand this to:

```
% echo men5jb /home/gps/men5jb
```

```
% ls /home/gps/men5jb
```

**Activity 3.2** Some variables may be set or changed by you. You may have an entry for the prompt variable For example:

```
prompt gps% (!)
```

To alter this (or to include it if it doesn't exist) type:

```
% set prompt = "what next % "
```

When you press <Enter> your prompt (invitation to type) should appear as:

```
what next %
```

The double quotes were required since the prompt we wanted included spaces.

**Activity 3.3** Previously defined variables may be used to define new ones. Ensure that the variable user is defined by typing:

```
% echo $user
```

Your own user name should get returned e.g. men5jb. If this does not happen define this variable yourself with:

```
% set user = user_name
```

Now define the prompt to include this variable:

```
% set prompt = "yes $user % (\!) "
```

Within the prompt `!` is replaced with the command number. The presence of the backslash will be explained in Task 6.

**Activity 3.4** You can make up variable names yourself and give them the value you chose. For example:

```
% set me = "The Green Knight "
% echo welcome $me
```

**Activity 3.5** There is another set of variables that exist which define the environment you are working in. To see these type the command:

```
% printenv
```

You will find that some of these have the same definitions as the shell variables of the same name but in a different case. This is because the environment variables are set up at login time from the shell variables so `HOME` takes the value of `home`, `USER` the value of `user`, etc.

**Note** Any variables you set up at this stage will be lost when you log out. If you want your own variables to be activated when you login you must place the definitions in the file `.cshrc` in your home directory.

# Task 4 Using The History Facility

**Objective** To display previously used Unix commands and use the recall facility to edit and re-run commands.

**Comments** Using the history facility can help you to keep a record of your actions and the recall facility can save you having to retype long command lines.

---

**Activity 4.1** Type the **`history`** command:

```
% history
```

Unix will display output something similar to the following:

```
21 set
22 echo $user $home
23 ls $ home
24 set prompt = " what next % "
25 echo $user
26 set prompt = "yes $user % (!) "
27 set me = " The Green Knight "
28 echo welcome $me
29 history
```

This is a record of previously entered commands.

**Activity 4.2** A previously used command can be re-run using ! *meta character*. The command **`!!`** will re-run the previous command and **`!`*n*** will rerun the *n*th command. To verify this type:

```
% !!
```

and:

```
% !23
```

Previous commands can also be recalled in a number of other ways, including (a) using a *relative* event number, such as:

```
% !-3
```

will run the third previous command; and (b) using character identification:

```
% !wh
```

which will run command 20 in the example given above because the shell looks backwards through the history list until it finds the first command whose initial characters match those supplied in the recall command.

**Activity 4.3** If a mistake is made in typing a command, it can be recalled and edited. For example type the following command (including the spelling mistake):

```
% cat /etc/princctap
```

To recall the last command and correct the spelling error type:

```
% !!:s/cct/tc/
```

The **:** separates the command specifier **!!** and the substitution command **s**, which replaces the first occurrence of *cct* with the characters *tc*

You can also add to the end of a previously used command. For example to rerun the **cat** command and pipe it to the **wc** command type:

```
% !c | wc
```

**Activity 4.4** Arguments from the last command can be extracted and used in the current command. For example **!$** denotes the last argument of the last command. Type the commands:

```
% cat /etc/printcap
% more !$
```

The **!$** is substituted by `/etc/printcap` from the previous command.

The command argument **!*** will be substituted by all the arguments of the previous command. The following commands will create three (disposable) files called `file1, file2` and `file3`, list out those files then delete them. Type:

```
% ls > file1
% who > file2
% finger > file3
% cat file1 file2 file3
% rm -i !*
```

The last command translates into **rm file1 file2 file3** deleting the files from your directory. Check to see that the files are gone by typing:

```
% ls file*
```

**WARNING** The command **rm !*** without the **-i** given above can be dangerous. If you inadvertently type the command **rm ! *** (with a space between the ! and * characters) the * wildcard character will be expanded to include *all* files in the directory. The **-i** ensures that you are prompted before each deletion.

# Task 5 Using Aliases

**Objective** To use the **alias** facility to abbreviate commonly used commands.

**Comments** If you use a few commands very often you may find it useful to define aliases for them. Aliases can also be used to give an alternative name to a command e.g. the **mv** (move) command can be given an alias **rename**.

---

**Activity 5.1** Give the **clear** command to the Unix prompt and verify that this command clears your screen:

```
% clear
```

If **clear** is a command that you use often you could save yourself time by typing **c** instead of **clear**. This can be done using the **alias** command:

```
% alias c clear
```

To test the result type: `% c`

You should now get the same action as with the **clear** command.

**Activity 5.2** You can look at the aliases you have defined by typing **alias** on its own.

```
% alias
```

You should see a list of aliases; most of them will have been set up for you automatically, but you should also see a line containing `c clear`.

**Activity 5.3** Aliases can be removed using **unalias** but at this stage you are strongly advised not to remove any of those set up by the system.

Try removing the one you have just defined by typing the command:

```
% unalias c
```

and check that it has gone with **alias**.

**Activity 5.4** Shell variables can be used in alias definitions. Define an alias **seeme** to return some information about the session you are working on. Type:

```
% alias seeme finger $user
% seeme
```

**Activity 5.5** Aliases can be combined with history information enabling parameters to be passed to the command. Type:

```
% alias toppage head -25 \!\$
% toppage /etc/hosts
```

Note that **head -n *filename*** will print the top *n* lines of the given file.

When you type **toppage /etc/hosts** the shell expands this to **head -25 !$** and the **!$** is replaced by `/etc/printcap` (the last argument of the previous command). In effect **toppage** has been defined as the top 25 lines of the file you give when you call the command.

# Task 6 Understanding Quotes

**Objective** To see how the shell interprets text enclosed in quotes (', " and ` quotes) and how the backslash can be used to escape metacharacters.

**Comments** In the previous exercises you used *metacharacters* (characters such as **>, |** and **$** that have a special meaning). When the shell sees one on a command line it takes some special action. Sometimes you want to use a metacharacter as input to a command without the shell carrying out the special action. Quotes and the \ character allow you to do this.

---

**Activity 6.1** Type the commands:

```
% echo $home $shell
% echo 'The value of $home and $shell is ' $home
$shell
% echo how many words and characters in this line |
wc
% echo 'how many words and characters in this line
| wc'
```

Enclosing the characters in **'** quotes tells the shell to ignore the special meaning of the metacharacters.

**Activity 6.2** The backslash (\) character says ignore the special meaning of the following single character: the second and fourth line above could have been written:

```
% echo The value of \$home and \$shell is  $home
$shell
% echo how many words and characters in this line
\| wc
```

Here the **$** and **|** metacharacters have been *escaped* using the backslash.

**Activity 6.3** Double quotes serve a similar function to single quotes but this time *not all* the special characters are ignored. In particular the **$**, back quotes**`**, backslash ****, **!** and carriage return retain their special significance.

```
% echo Your home directory is $home
% echo Your home directory is $home > temp.file
% cat temp.file
% echo "Your home directory is $home > temp.file"
```

In the last case **$home** is expanded but **>** loses its special meaning. Type in the following and check that you understand the output.

```
% set x = 7
% echo The value contained in \$x is \"$x\"
```

Refer back to Activities 3.3 and 5.5 to see how the **** was used.

**Activity 6.4** The back quote is used for a different purpose than other metacharacters:

```
% date
% echo Today\'s date is date
% echo Today\'s date is `date`
```

Anything enclosed in back quotes is evaluated and inserted at the point it appears in the command. This is known as *command substitution*.

```
% set welcome = "Welcome to Unix $user Today is
`date` \
You have `ls | wc -l` files "
% echo $welcome
```

The carriage return at the end of the line had to be *escaped* or the shell would think the command line had ended after **`date`**.

# Task 7 Shell Programming

**Objective** To write a program using shell commands and run it, and to find out how to use arguments within a shell script.

**Comments** Shell commands can be stored in a file to be executed when required. Such files are known as *shell scripts*.

---

**Activity 7.1** Using a Unix editor create a shell script called `summary` containing the following lines:

```
#!/bin/csh -f
#
ls -lta
who
echo "Total Number Of Users Logged in is  "
who | wc -l
finger $user
```

**Note** If you have problems obtaining a character type the £ character instead

Apart from the first, all the other lines contain shell commands that should be reasonably familiar to you. To execute this shell script type:

% **csh summary**

**Activity 7.2** It would be nice to be able to execute this script simply by typing the name of it, i.e. `summary`, however if you try this you will probably get the error message `Permission Denied`. This is because the system does not know that it is an executable file. Type:

```
% ls -lt summary
```

You will get returned something like `-rw-------  summary.` The characters **rw** signify that you can read and write to the file but not execute it. To make it executable type:

```
% chmod  u+x  summary
% ls -lt summary
```

The **chmod** (change mode) command has added execute (**x**) status for the user (you) to the file `summary`. Try running it again using the name of the script:

% **summary**

**Activity 7.3** Arguments can be passed to the shell script in the command line.

The arguments are referred to by **$1, $2, $3,** etc (or **$argv[1], $argv[2], $argv[3]**) within the script.

Create a shell script called `rename` containing the following:

```
#!/bin/csh -f
echo Contents of File are
cat $1
mv $1 $2
echo $1 is now renamed to $2
echo command $0 completed
```

Run this with the file created above:

```
% chmod u+x rename
% rename summary sum
```

Note that **$0** translates to the name of the command.

**Activity 7.4** The number of arguments passed to a script is represented by **$#argv** so in the above example you could check that the correct number of arguments were passed by including as the second line of the script:

```
if ($#argv != 2) exit
```

Note that **!=** means not equal to.

**Activity 7.5** Arguments may also be passed while the script is running using the characters $<. This allows for the user to be prompted for the required input. Edit the above file to include the following lines:

```
#!/bin/csh -f
echo -n Please enter name of File to be renamed:
set file = $<
echo -n Please enter new name required for $file
set newfile = $<
echo Contents of File are
cat $file
mv $file $newfile
echo $file is now renamed to $newfile
echo command $0 completed
```

Run this by typing **rename** without giving any arguments. This time you will be prompted for the filenames; use **sum** and **summary** to return the file to its original name.

The `-n` on the lines containing the **echo** command ensures that the cursor remains on the prompt line.

**Activity 7.6** Input can also be supplied from within the script file itself. Before this is demonstrated create the following two files.

1 A shell script file named `address`

```
#!/bin/csh -f
echo "The Address of $1 is :- "
grep $1 address.list
```

2 A data file containing a list of names and addresses called `address.list`

```
Jane Brown    4 Ash Road
John Smith    10 Cedar Avenue
Mary Davies   125 Wood Lane
Peter Jones   65 Birch Crescent
Mary Smith    145 Oak Drive
```

Run the shell script by typing:

```
% chmod u+x address
% address Mary
```

You should get returned the line from the address.list file giving the address of Mary Davies and Mary Smith.

The command **grep** will search a file (in this case `address.list`) for any lines that contain the required character string (Mary) and print them out.

**Activity 7.7** The *here document* facility can be used to do the same job. In order to use this facility you should first type the commands:

```
% cat address address.list > address2
```

to combine the two files into a new file called `address2`. Then edit this file to include the text given in bold (**`<< fin`** and **`fin`**):

```
#!/bin/csh -f
echo "The Address of $1 is :- "
grep "$1" << fin
Jane Brown  4 Ash Road
John Smith   10 Cedar Avenue
Mary Davies  125 Wood Lane
Peter Jones  65 Birch Crescent
Mary Smith   145 Oak Drive
fin
#
```

The **`<<`** metacharacter indicates the presence of a 'here document' i.e. *take all input from the following lines until the terminator is reached*. The terminator in this case being the string **`fin`** Now type:

```
% chmod u+x address2
% address2 "Mary Smith"
```

This time you should get returned only the address of Mary Smith.

Note that the use of the double quotes allows you to pass a space within the search string.

**Note** The first line of each of the shell scripts you have written:

```
#!/bin/csh -f
```

is very important. It tells Unix that you want this script run under the C shell. If you do not include this line the script will be run under the Bourne shell and some of the commands you use may not be understood.

# Task 8 Managing Jobs

**Objective** To identify jobs and see how they can be suspended, restarted and cancelled.

**Comments** A user may have more than one job (or process) running at any time so it is important to know how to control the running of multiple processes.

---

**Activity 8.1** There are two commands you can use to enquire about the processes you are controlling: **ps** and **jobs**. Type the commands:

```
% ps
% jobs -l
```

The first command will return something like:

```
PID  TT   TIME  COMD
5031 p6   0:00  csh
5082 p6   0:00  ps
```

The second command should return nothing. This is because **ps** returns *all* processes, including the shell you are running and the **ps** process itself, while **jobs -l** lists the active jobs under your control excluding itself and the shell.

**Activity 8.2** Create the following file and name it `times_table`. This will be used to demonstrate the management of processes:

```
#!/bin/csh -f
set value = 1
while ($value != 4000)
set result = `expr $value \* $1`
echo $value times $1 equals $result
set value = `expr $value + 1`
end
```

Run the script by typing:

```
% chmod u+x times_table
% times_table 5
```

The script calculates the 5 times table. To stop this process type <Ctrl C>. Type **jobs -l** to check that the process is no longer there.

**Activity 8.3** A process can be suspended without being totally cancelled. Run the script again but this time type **<Ctrl Z>**. Then type:

```
% jobs -l
```

You should get returned something similar to :

```
[1]     +    1234  Stopped   times_table 5
```

To restart this process type:

```
% fg
```

(meaning foreground) and then cancel it all together with **<Ctrl C>**.

**Activity 8.4** Set several jobs off, suspend them, then restart in a different order. Type:

```
% times_table 2
<Ctrl Z>
% times_table 4
<Ctrl Z>
% times_table 6
<Ctrl Z>
% jobs -l
```

You should see something like:

```
[1]        7268  Stopped    times_table 2
[2]   -    7301  Stopped    times_table 4
[3]   +    7322  Stopped    times_table 6
```

The first column gives the job number and the third gives the *process identifier* (PID).

The + in the second column indicates the current job (most recent) and the - indicates the previous job. If you type **fg** (or **fg %+**) this will restart the current job.

The previous job may be restarted with **fg %-** The command **fg %***n* will restart job number *n*.

Try starting job 1 by typing:

```
% fg  %1
```

Then cancel it with **<Ctrl C>**

**Activity 8.5** Stopped jobs may be cancelled altogether using the **kill** command. This command can be used with the job number:

```
% kill  %2
```

or with the process identifier (PID):

**% kill *PID*** e.g. **% kill 7322**

Cancel one of your stopped jobs with the above command but keep one in the stopped state.

**Activity 8.6** If you try to logout while you have jobs stopped you will get a warning. Type:

```
% logout
```

The shell will display the message:

```
There are stopped jobs
```

and will not log you out. This is a warning to you in case you have forgotten about jobs you stopped. If you are not interested in the stopped jobs just type **logout** again, you will be logged off and the jobs will disappear. Otherwise you can restart the jobs and continue work.

Before you start the next task make sure all jobs have been killed.

**Note** Occasionally **kill *PID*** will fail to remove a job. If this happens type:

**% kill -9 *PID*** e.g. **% kill -9 7322**

# Task 9 Running Background Jobs (1)

**Objective** To find out how to start jobs in the background and control the running of them.

**Comments** Jobs can be run as background processes. This means that you can carry on doing interactive work while other jobs continue with their processing.

---

**Activity 9.1** Jobs may be run in the *background* by adding **&** to the end of the command.

Type the command:

```
% times_table 4  &
```

You should get returned:

```
[1]   PID
```

followed by the output from the script. This does not look much different from the output produced by the first part of Activity 8.2 but this process is actually running as a background process and you can type in other commands while it is running. Type:

```
% jobs -l
```

Mixed in with the output from the shell script you will get a line similar to:

```
[1]  +  PID   Running   times_table  4
```

The script output still appears on the screen because it is writing to *standard output* which, since we have not redirected anywhere, remains as the screen.

The background job may be cancelled either by using the **_kill** command:

```
% kill PID          e.g.          % kill 7322
```

or by bringing the process to the foreground with **fg** and cancelling it with **<Ctrl C>:**

```
% fg
<Ctrl C>
```

**Activity 9.2** Alter line 5 of your script file to the following line:

```
echo $value times $1 equals $result > $2
```

This will redirect the output to a file given by a second argument, thus preventing output from the background job from interfering from your interactive session.

Run three background jobs but this time redirect their output to three different files.

```
% times_table 5  output1 &
% times_table 8  output2 &
% times_table 3  output3 &
```

Check for yourself that these are running by typing **jobs -l.** Also look at the output files using **cat output1** etc. and see that their contents are changing. Note that only one line of output is stored in each of the output files (to prevent them from getting excessively large).

**Activity 9.3** Background jobs can be brought to the foreground using the **fg** command. Type:

```
% fg %1
```

You will find that you cannot type in other commands because there is now a job running in the foreground.

You could kill this job by pressing **<Ctrl C>** but for the present we want to keep it running in the background so type

**<Ctrl Z>**

```
% bg
```

**Activity 9.4** Background jobs can be suspended with the **stop** command which can be used with either the *PID* or *%job_number*. Type the following commands to stop jobs 1 and 2.

```
% stop %1
% stop PID          e.g.          % stop 7322
```

and check the output files 1 and 2 to see that they have stopped changing.

**Activity 9.5** Restart a suspended job in the background with the **bg** command. On its own this will restart the current job, or the job number may be supplied.

Type the following to restart the jobs above:

```
% bg %1
% bg %2
```

**Note** If a process running in the background gets to a point where it requires input from the terminal it will wait until it is brought to the foreground and the input is supplied.

# Task 10 Running Background Jobs (2)

**Objective** To monitor background jobs left running from another session.

**Comments** If you leave background jobs running when you logout it is important to remember them as they continue to use up resources. The jobs continue to run when you logout: you will not be warned as with stopped jobs.

---

**Activity 10.1** Check that you have at least one job *running* as a background process by typing **`jobs -l`** and make a note of its PID. Then logout:

```
% jobs -l
% logout
```

**Activity 10.2** Log back onto your machine and type:

```
% jobs -l
% ps
```

Neither command indicates the presence of the job you left running before you logged off: they only display processes started under the current shell.

Using the PID that you made a note of before logging off, type the command:

**`% ps -p`** ***`PID`*** e.g. **`% ps -p 5031`**

[ on suns use **`% ps`** ***`PID`*** e.g. **`% ps 5031`** (see note below)]

You should see the process.

**Activity 10.3** How do you find the process if you haven't noted its PID? Type:

```
% ps -ef
```

This asks for information on all processes being run by the machine (**e**) including those run by other users, printed out in full (**f**). You should see your background process somewhere in the list. To filter out those you are not interested in use **`grep`** to select only those processes owned by you:

**`% ps -ef | grep`** ***`user_id`***

This will produce a shorter list and your process should be easily identified.

The ? under the tty line column indicates that the process is running independently and not being controlled by a terminal, unlike the processes running under your current shell.

**Activity 10.4** Kill this process and check that it has gone by typing:

```
% kill PID
% ps -ef | grep user_id
```

**Note** Due to the fact that some of our machines run a different version of the operating system, this command differs slightly from one machine to another. For full details of the **`ps`** command type **`man ps`**.